

컴퓨터그래픽스

2017학년 1학기
김준호

국민대학교 소프트웨어학부

- Affine Space & Homogeneous Coordinates
- Linear transformations
- Model transformations
- View transformations

Transformations

Affine Space & Homogeneous Coordinates

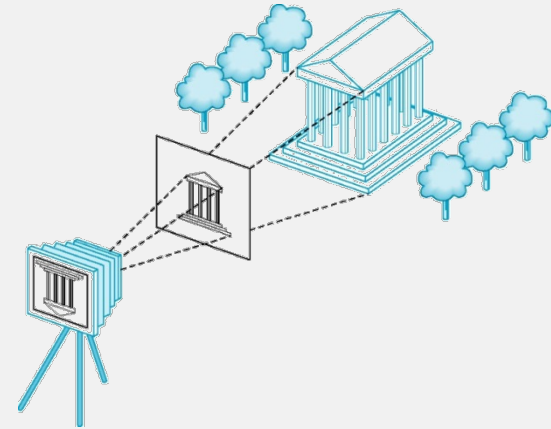
OpenGL ES uses Homogeneous coordinate System

- Graphics cards support homogeneous coordinates
 - 4x1 vectors for 3D points & 3D vectors

$$\mathbf{p} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{bmatrix} \qquad \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ 0 \end{bmatrix}$$

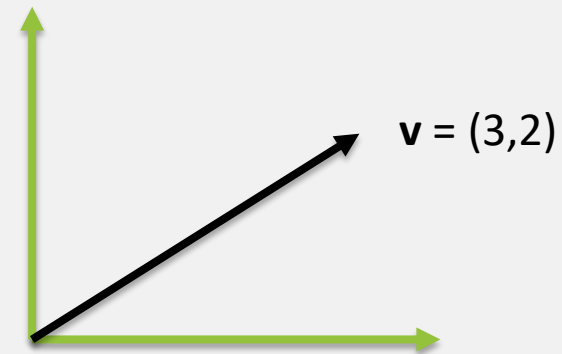
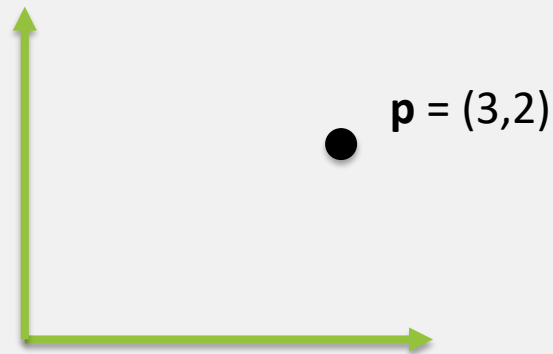
- In general, the following holds (if $a \neq 0$, $w \neq 0$)
 - It is related to projective geometry

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} ax \\ ay \\ az \\ aw \end{bmatrix} = \begin{bmatrix} x/w \\ y/w \\ z/w \\ 1 \end{bmatrix}$$



Affine Space

- The affine space contains three types of object
 - Scalars
 - Vectors
 - Points
- Why affine space?
 - We need to clearly distinguish the concepts of vectors and points
 - There was no strong differences in vectors and points, especially in high school



Affine Space

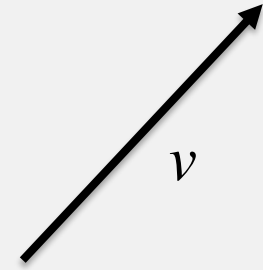
- Scalar \oplus Vector \oplus Point
- Operations
 - Vector-vector addition
 - Scalar-vector multiplication
 - Point-vector addition
 - Scalar-scalar operations
- For any point define
 - $1 \bullet \mathbf{P} = \mathbf{P}$
 - $0 \bullet \mathbf{P} = \mathbf{0}$ (zero vector)

Scalars

- Scalars represent the concept of quantity
 - Ex) 7, 3.14, -1, ...
 - Combined with two basic operations
 - Addition
 - Multiplication
- Scalars alone have no geometric properties

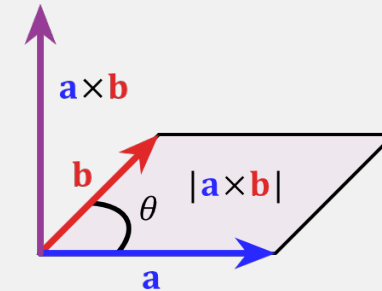
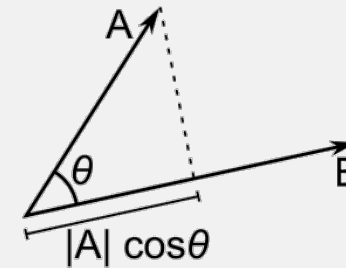
Vectors

- A vector is a quantity of two attributes
 - Direction
 - Magnitude
- A vector usually represented with a directed line segment
- Examples include
 - Force
 - Velocity



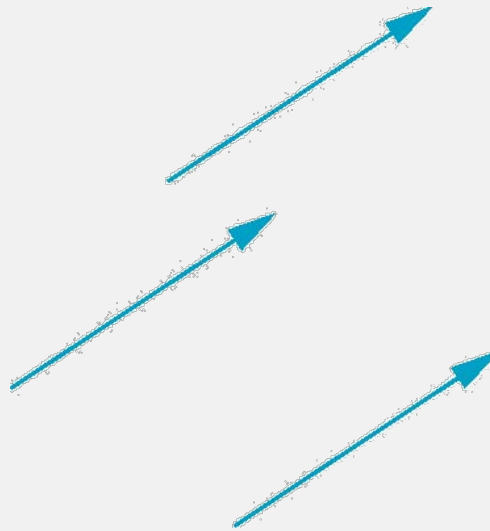
Vector Operations

- scalar * vector
 - Ex) $2\mathbf{v}$, $-\mathbf{v}$, ...
 - Result: a vector
- vector + vector
 - Ex) $\mathbf{u} + \mathbf{w}$
 - Result: a vector
- Products
 - Dot product
 - Ex) $\mathbf{u} \cdot \mathbf{w}$
 - Result: a scalar
 - Physical meaning: amount of projection
 - Cross product
 - Ex) $\mathbf{u} \times \mathbf{w}$
 - Result: a vector
 - Physical meaning: amount of rotation



Vectors Lack Point

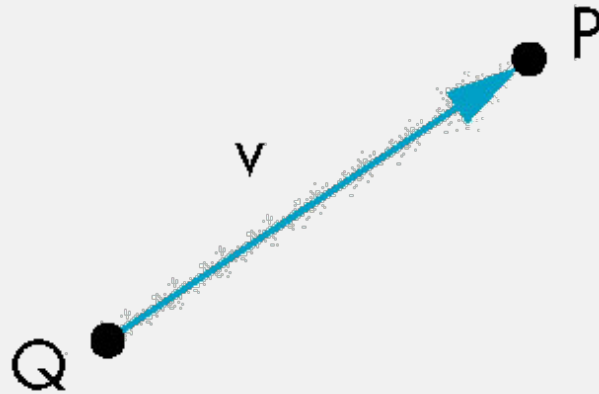
- The following vectors are identical
 - Same length and magnitude



- (Scalars + Vectors) are insufficient for representing geometry
 - Need points

Points

- Location in space
- Operations allowed between points and vectors
 - Point-point subtraction yields a vector
 - $\mathbf{P} - \mathbf{Q} = \mathbf{v}$
 - $\mathbf{P} + \mathbf{Q} \rightarrow$ no physical meaning
 - Equivalent to point-vector addition
 - $\mathbf{Q} + \mathbf{v} = \mathbf{P}$

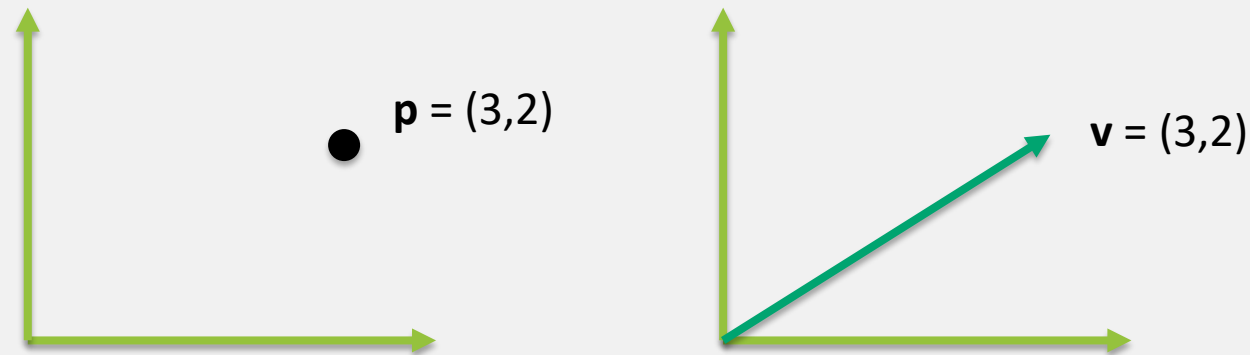


$$\mathbf{v} = \mathbf{P} - \mathbf{Q}$$

$$\mathbf{P} = \mathbf{v} + \mathbf{Q}$$

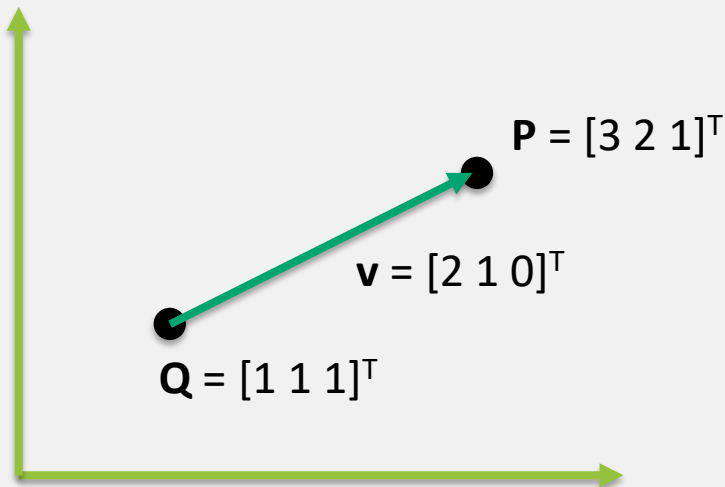
Homogeneous Coordinates

- Homogeneous coordinate systems clearly distinguish the concepts of vectors and points
 - n D point is represented with a $(n+1)$ D vector, whose last component is 1
 - n D vector is represented with a $(n+1)$ D vector, whose last component is 0
- Example) 2D points, 2D vectors \rightarrow 3D vectors
 - $\mathbf{p} = [3 \ 2 \ 1]^T$
 - $\mathbf{v} = [3 \ 2 \ 0]^T$



Homogeneous Coordinates

- Homogeneous coordinates hold the operations of affine space
 - E.g.) $\mathbf{P} = [3 \ 2 \ \mathbf{1}]^T$, $\mathbf{Q} = [1 \ 1 \ \mathbf{1}]^T$, $\mathbf{v} = [2 \ 1 \ \mathbf{0}]^T$
 - $\mathbf{v} = \mathbf{P} - \mathbf{Q} = [3 \ 2 \ \mathbf{1}]^T - [1 \ 1 \ \mathbf{1}]^T = [2 \ 1 \ \mathbf{0}]^T$
 - $\mathbf{P} = \mathbf{Q} + \mathbf{v} = [1 \ 1 \ \mathbf{1}]^T + [2 \ 1 \ \mathbf{0}]^T = [3 \ 2 \ \mathbf{1}]^T$
 - $2\mathbf{v}$
 - $3\mathbf{Q}$



Why Homogeneous Coordinate?

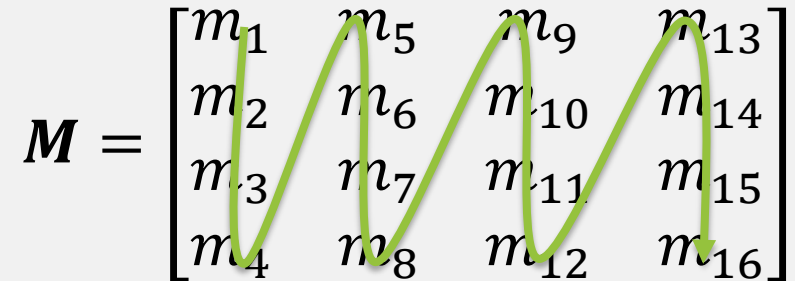
- Graphics cards support homogeneous coordinates

- 4x1 vectors for 3D points & 3D vectors

$$\mathbf{p} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{bmatrix} \qquad \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ 0 \end{bmatrix}$$

- 4x4 matrices for

- GL_MODELVIEW_MATRIX, GL_PROJECTION_MATRIX, GL_TEXTURE_MATRIX
- Note: OpenGL ES represents a matrix in the column-major way

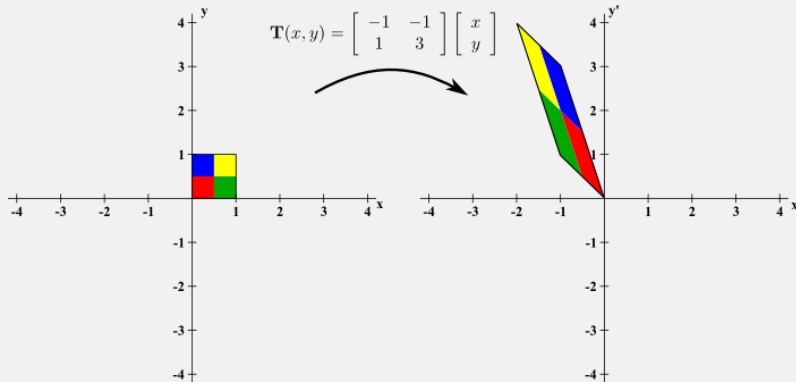
$$\mathbf{M} = \begin{bmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{bmatrix}$$


Linear Transformations

Transformation (변환)

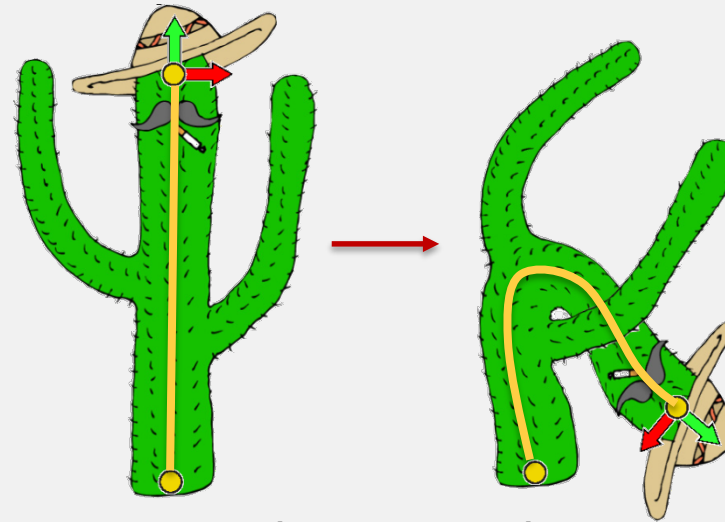
- In computer graphics, transformation refers to
 - Change of shape
 - Linear transformation: line to line
 - Non-linear transformation: line to curve

Linear transformation



http://mathinsight.org/determinant_linear_transformation

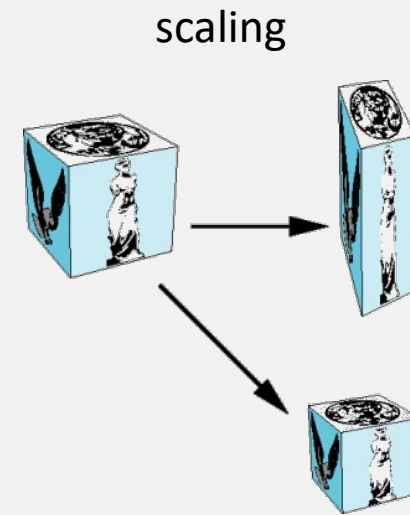
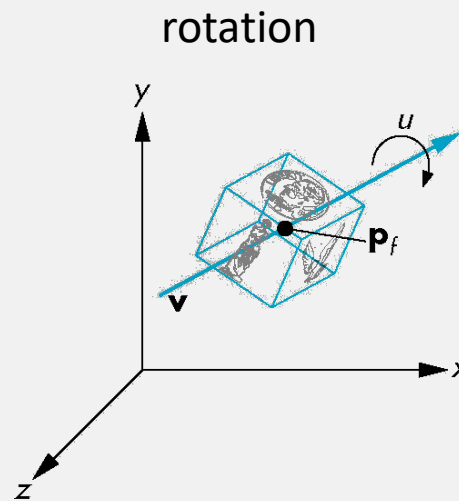
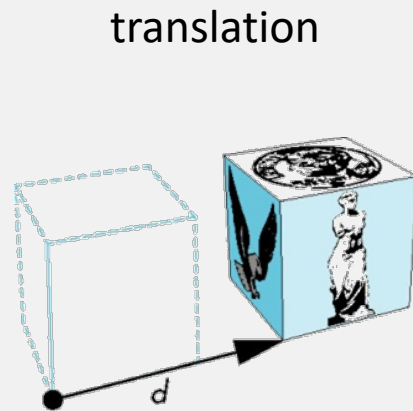
Non-linear transformation



[Jacobson et al. 2012]

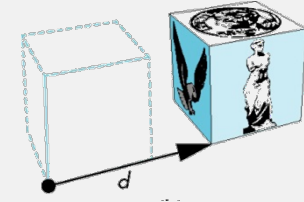
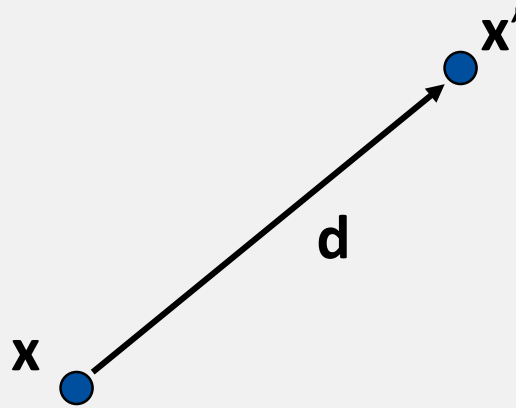
Linear Transformation (선형 변환)

- It can be represented as linear a matrix
- Standard transformation
 - Translation / Rotation / Scaling
- Composition of linear transformation is linear



Translation (이동)

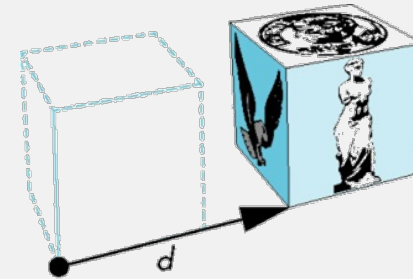
- Move (translate, displace) a point to a new location
 - Translation of an object: every point displaced by same vector



- Displacement determined by a vector d
 - Three degrees of freedom for d , in 3D case: $d = (d_x, d_y, d_z)$
 - $x' = x + d$

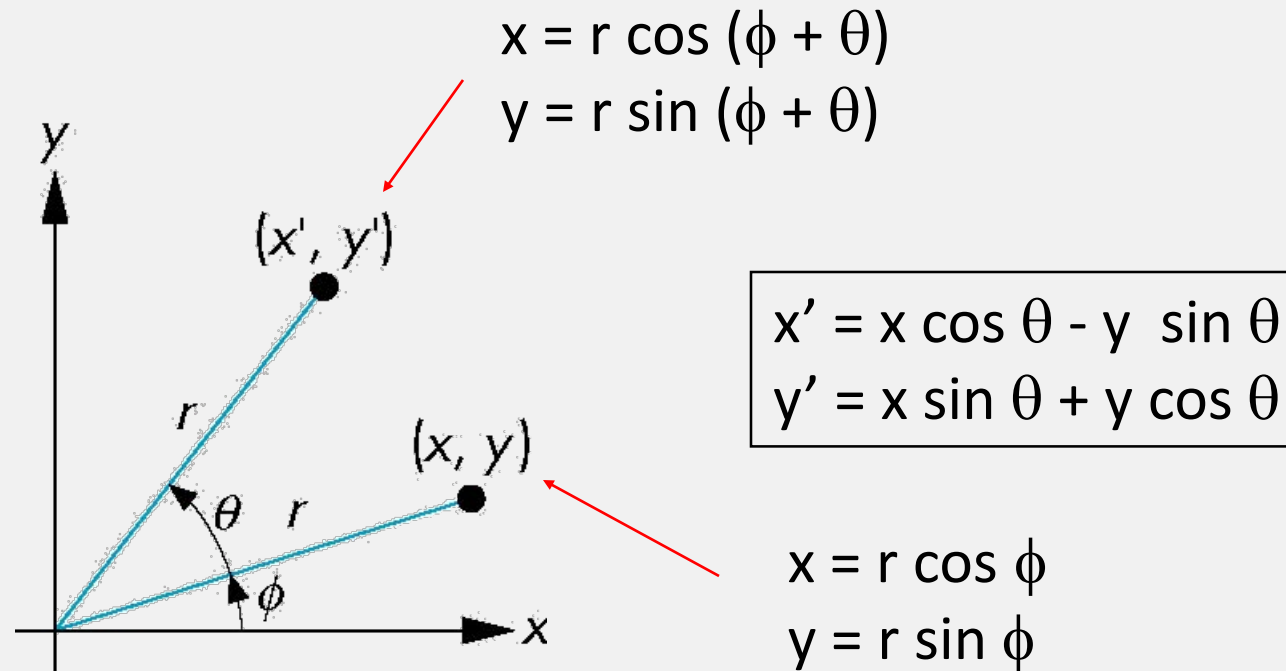
Translation (이동)

- Translation matrix **T**
 - Translation can also be expressed by using a 4x4 matrix **T** in homogeneous coordinates
 - $\mathbf{x}' = \mathbf{T}\mathbf{x}$
 - $\mathbf{T} = \mathbf{T}(d_x, d_y, d_z) =$
$$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
 - [`glTranslate`](#)(d_x, d_y, d_z)
 - This form is better for H/W implementation



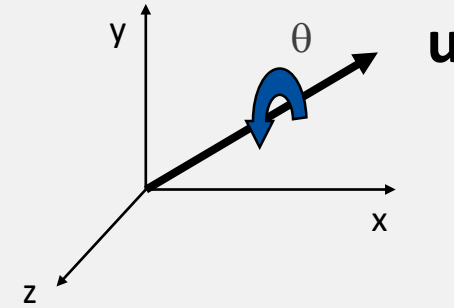
Rotation (회전)

- Consider rotation about the origin by θ degrees
 - Radius stays the same, angle increases by θ



Rotation (회전)

- General rotation about the **origin**
 - A rotation by θ rotation an arbitrary axis \mathbf{u}
 - [Quaternion](#) can express general rotation



- $\mathbf{x}' = \mathbf{R}_u(\theta)\mathbf{x}$

- $\mathbf{R}_u(\theta) =$

$$\begin{bmatrix} \cos \theta + u_x^2(1 - \cos \theta) & u_x u_y(1 - \cos \theta) - u_z \sin \theta & u_x u_z(1 - \cos \theta) + u_y \sin \theta & 0 \\ u_y u_x(1 - \cos \theta) + u_z \sin \theta & \cos \theta + u_y^2(1 - \cos \theta) & u_y u_z(1 - \cos \theta) - u_x \sin \theta & 0 \\ u_z u_x(1 - \cos \theta) - u_y \sin \theta & u_z u_y(1 - \cos \theta) + u_x \sin \theta & \cos \theta + u_z^2(1 - \cos \theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

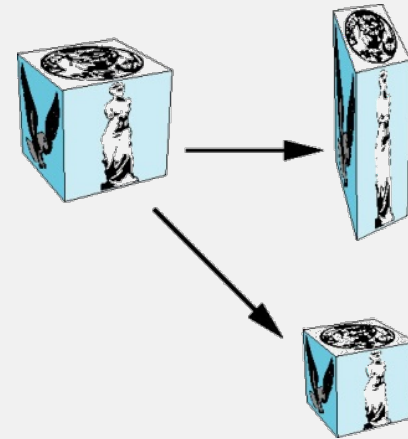
- [glRotate](#)(θ, u_x, u_y, u_z)
 - $\mathbf{u} = (u_x, u_y, u_z)$

Scaling (확대 축소)

- Scaling matrix **S**
 - Expand or contract along each axis (fixed point of **origin**)
 - $\mathbf{x}' = \mathbf{S}\mathbf{x}$

- $\mathbf{S} = \mathbf{S}(s_x, s_y, s_z) =$

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



- [`glScale`](#)(s_x, s_y, s_z)

Model Transformations

Composite Transformation (합성 변환)

- We can composite transformation by multiplying matrices of rotation, translation, and scaling.

– Example:

1) Uniformly scale 2x:

$$S = S(2,2,2)$$

2) Rotate -30 degrees by +z axis:

$$R = R_z(-30)$$

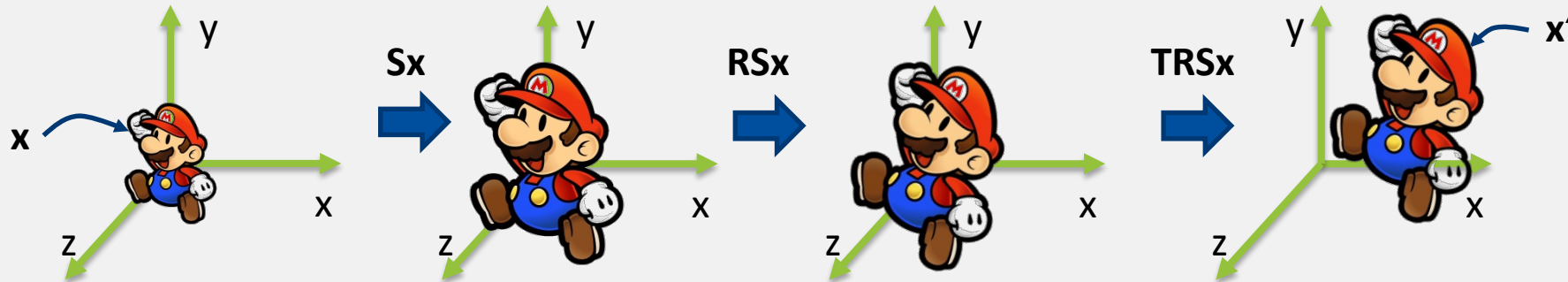
3) Translate by (2, 1, 0):

$$T = T(2,1,0)$$

- $\mathbf{x}' = T(R(S\mathbf{p})) = TRS\mathbf{x}$

– \mathbf{x} : original object

– \mathbf{x}' : transformed object



Composite Transformation (합성 변환)

- We can composite transformation by multiplying matrices of rotation, translation, and scaling.

– Example:

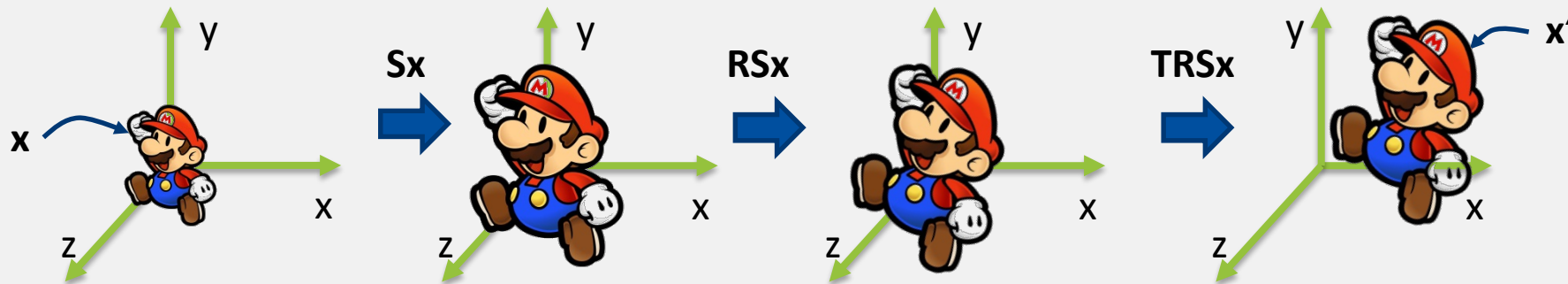
- 1) Uniformly scale 2x:
- 2) Rotate -30 degrees by +z axis:
- 3) Translate by (2, 1, 0):

- $\mathbf{x}' = \mathbf{T}(\mathbf{R}(\mathbf{S}\mathbf{x})) = \mathbf{TRSx}$

– \mathbf{x} : original object

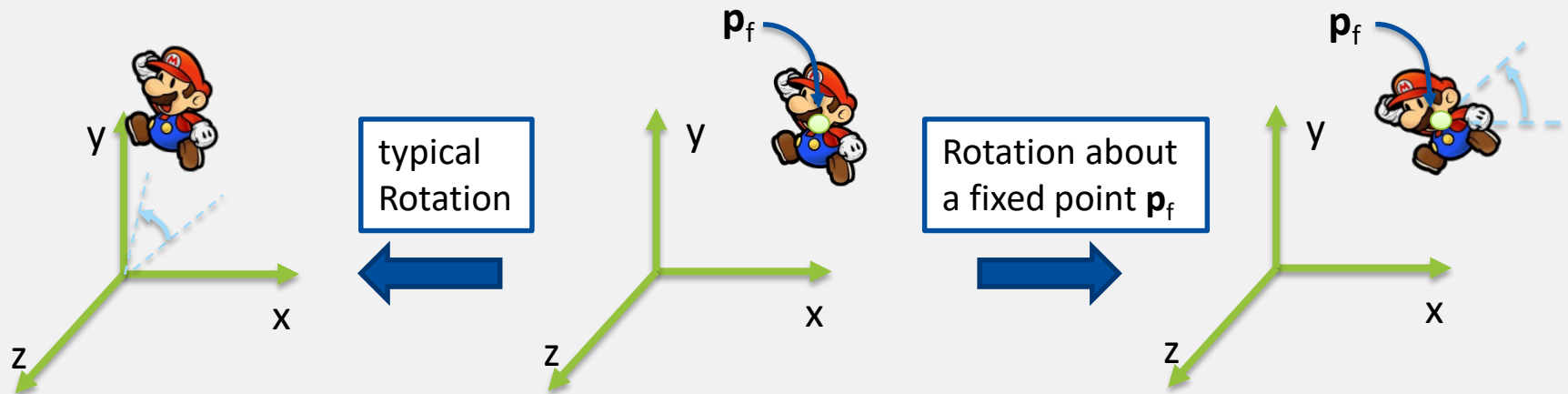
– \mathbf{x}' : transformed object

```
glTranslatef(2, 1, 0);      // T
glRotatef(-30, 0, 0, 1);   // R
glScalef(2, 2, 2);         // S
glDrawElements(...);      // x
```



Rotation about a Fixed Point other than the Origin

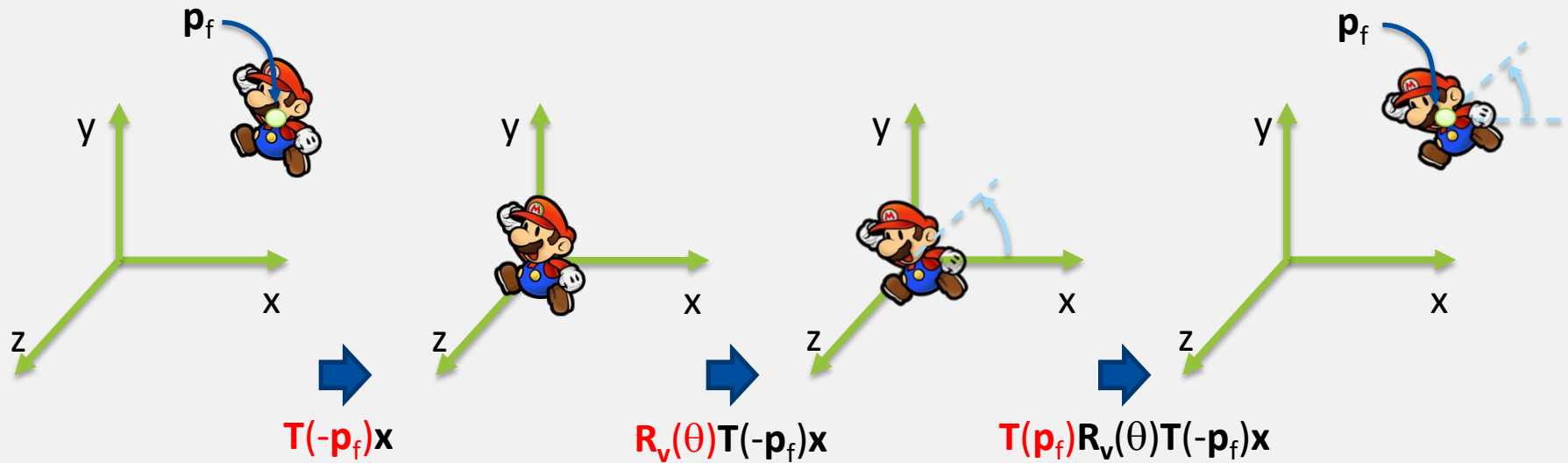
- Rotation: originally, a fixed point is the origin
 - Example
 - Rotate 30 degrees by +z axis: $R = R_z(30)$
- But, rotation about a general fixed point \mathbf{p}_f is necessary, in general



Rotation about a Fixed Point other than the Origin

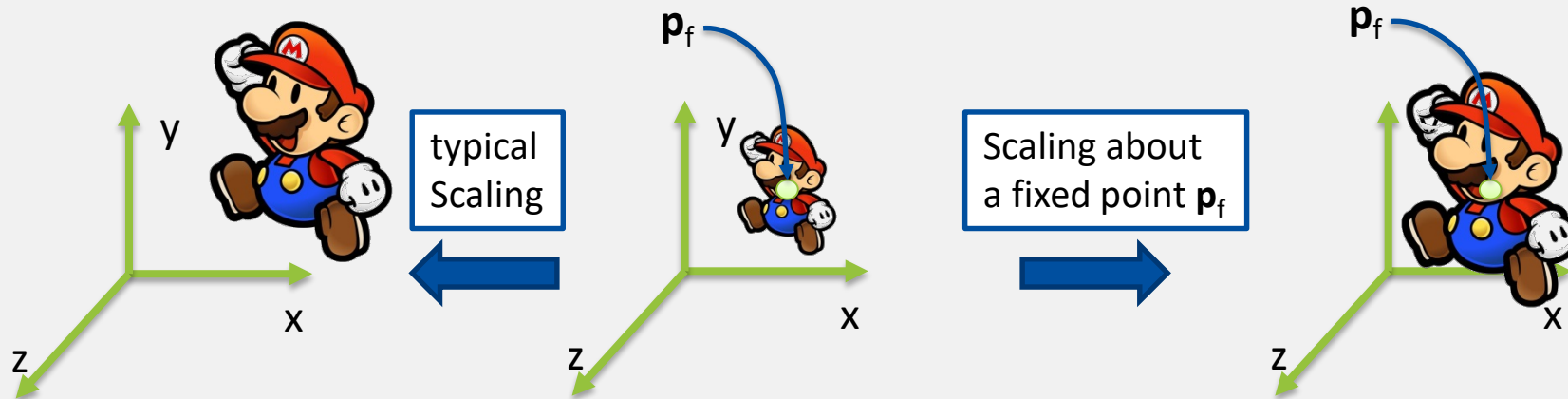
- Composite transformation
 - Move fixed point to origin
 - Rotate
 - Move fixed point back
- $\mathbf{x}' = \mathbf{T}(\mathbf{p}_f)\mathbf{R}_v(\theta)\mathbf{T}(-\mathbf{p}_f)\mathbf{x}$

```
glTranslate(pfx, pfy, pfz);    //  $\mathbf{T}(\mathbf{p}_f)$   
glRotate(theta, vx, vy, vz);  //  $\mathbf{R}_v(\theta)$   
glTranslate(-pfx, -pfy, -pfz); //  $\mathbf{T}(-\mathbf{p}_f)$   
glDrawElements(...);         //  $\mathbf{x}$ 
```



Scale about a Fixed Point other than the Origin

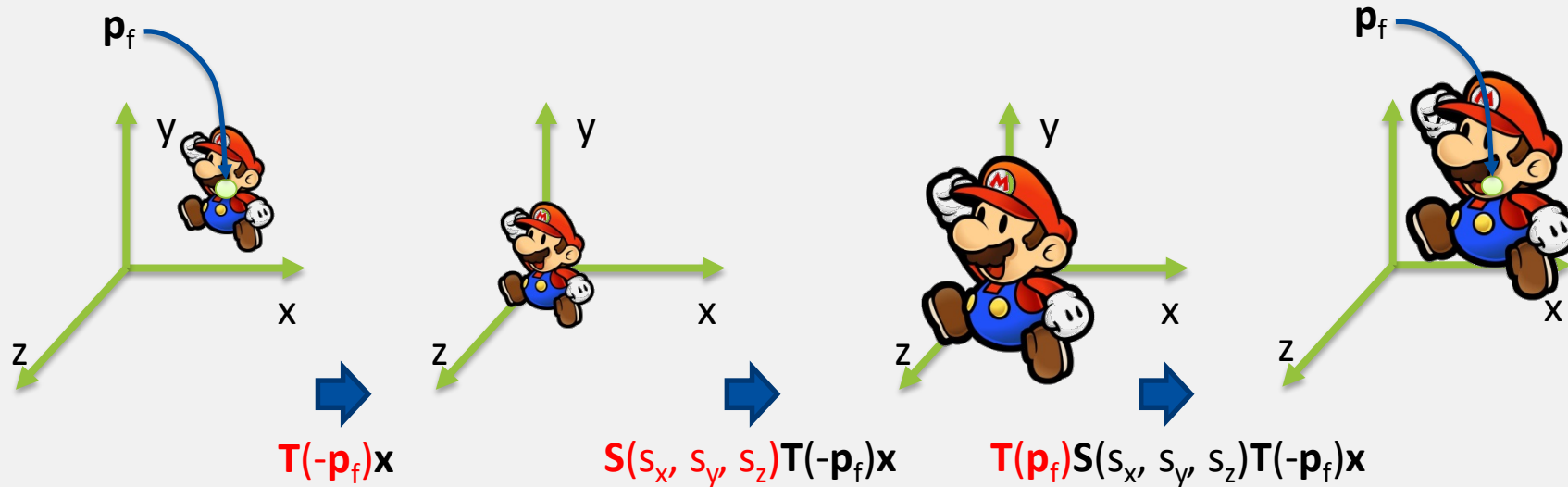
- Scaling: originally, a fixed point is the origin
 - Example
 - Uniformly scale 2x: $S(2,2,2)$
- But, scaling about a general fixed point \mathbf{p}_f is necessary, in general



Scale about a Fixed Point other than the Origin

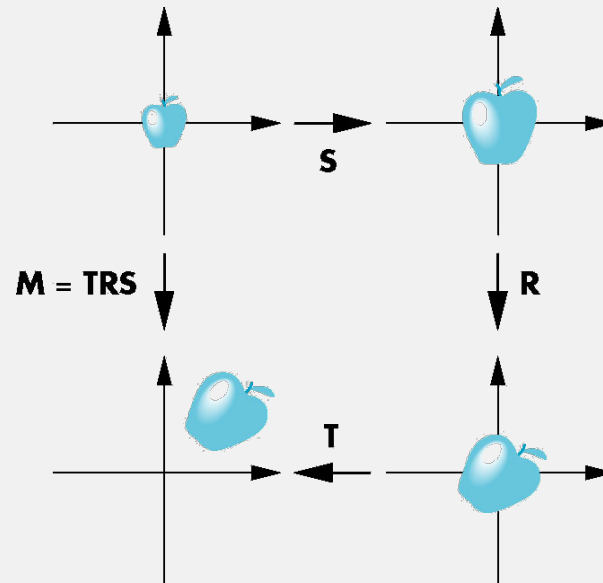
- Composite transformation
 - Move fixed point to origin
 - Scale
 - Move fixed point back
- $\mathbf{x}' = \mathbf{T}(\mathbf{p}_f)\mathbf{S}(s_x, s_y, s_z)\mathbf{T}(-\mathbf{p}_f)\mathbf{x}$

```
glTranslate(pfx, pfy, pfz);    //  $\mathbf{T}(\mathbf{p}_f)$   
glScale(sx, sy, sz);          //  $\mathbf{S}(s_x, s_y, s_z)$   
glTranslate(-pfx, -pfy, -pfz); //  $\mathbf{T}(-\mathbf{p}_f)$   
glDrawElements();             //  $\mathbf{x}$ 
```



Instancing

- In modeling, we often start with a simple object centered at the origin, oriented with the axis, and at a standard size
- We apply an instance transformation to its vertices to
 - Scale
 - Orient
 - Locate

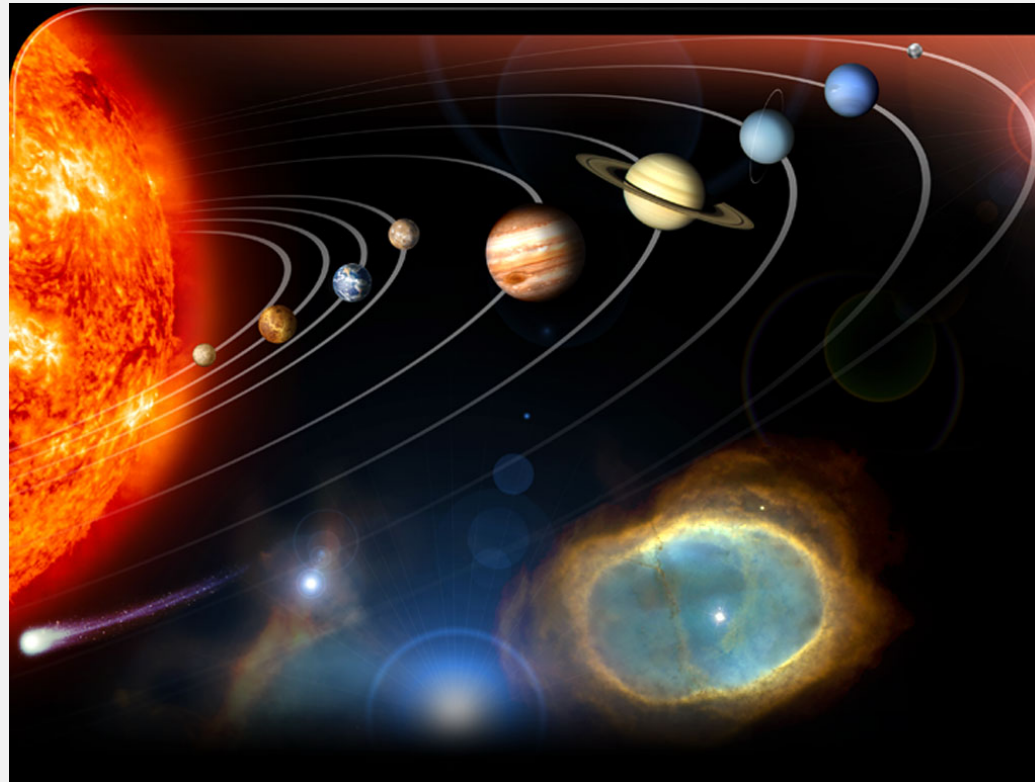


Advanced Transformation

Transformations for Hierarchical Objects

Solar System

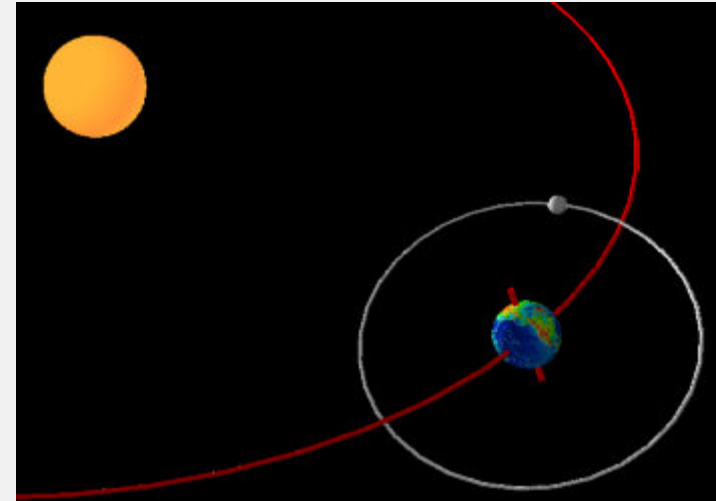
- An example of hierarchical objects
 - How can we design transformations for hierarchical objects



([video](#), [youtube](#))

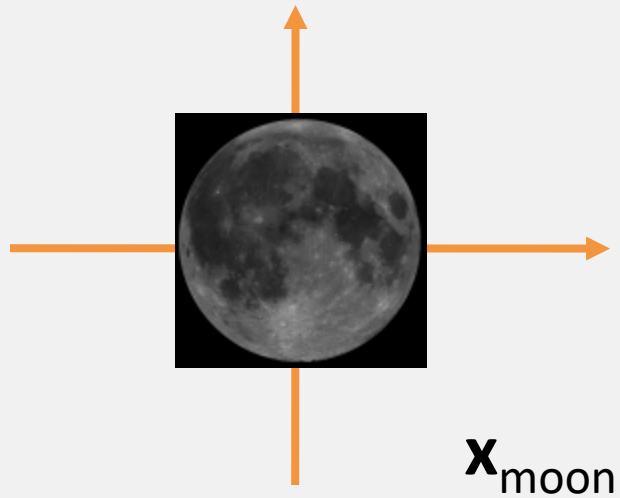
Sun – Earth – Moon

- Sun
- Earth
 - rotating itself
 - orbiting around the sun
- Moon
 - rotating itself
 - orbiting around the earth



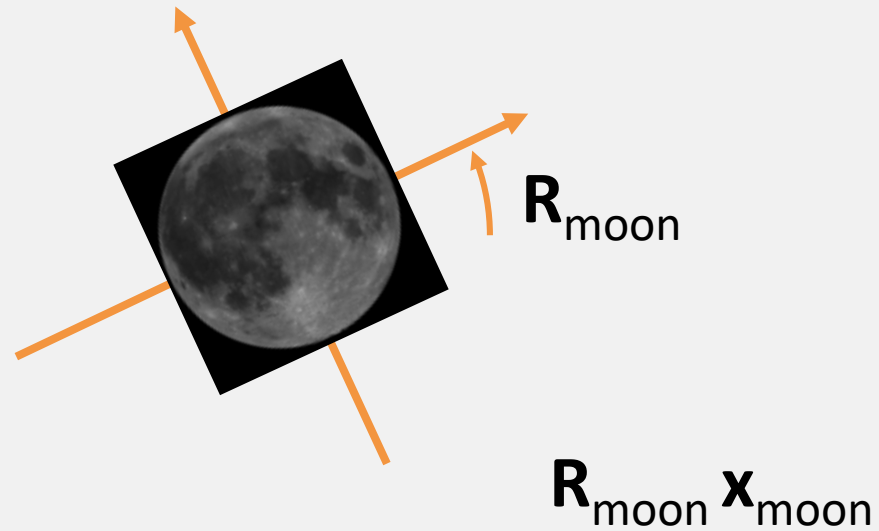
Hierarchical Transformation

- Moon
 - Modeling the moon



Hierarchical Transformation

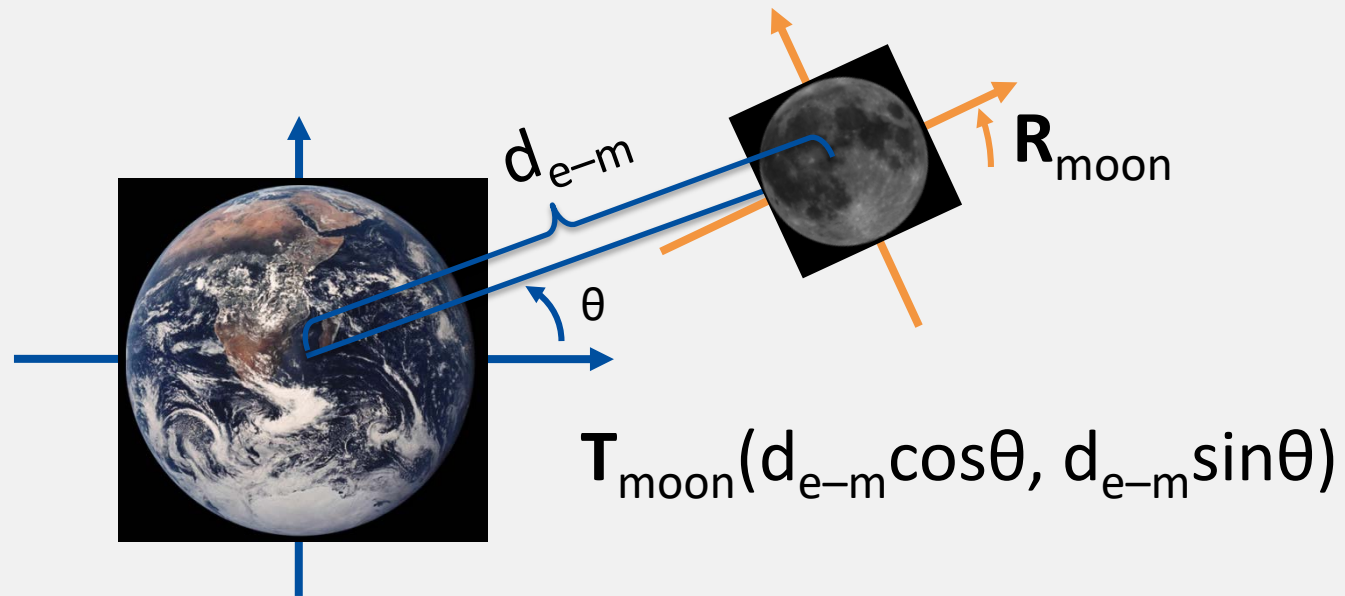
- Moon
 - Modeling the moon
 - Rotating itself



Hierarchical Transformation

- Earth – Moon
 - The moon is orbiting around the earth

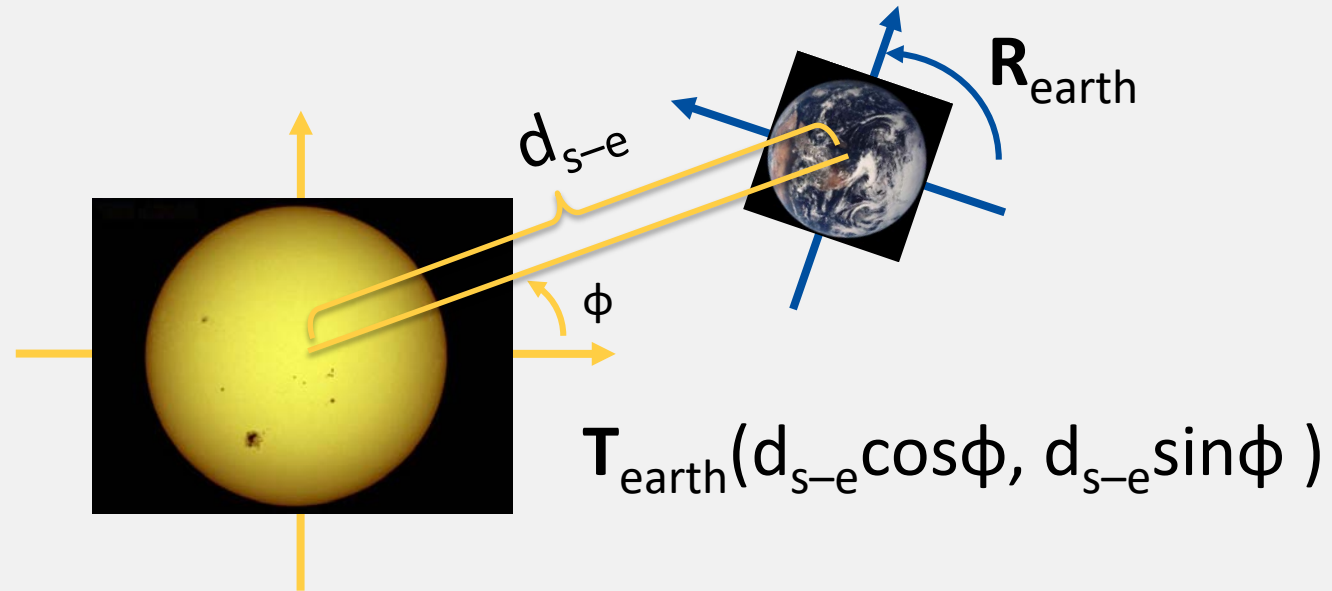
$$\mathbf{x}_{\text{earth}} = \mathbf{T}_{\text{moon}}(d_{\text{e-m}} \cos \theta, d_{\text{e-m}} \sin \theta) \mathbf{R}_{\text{moon}} \mathbf{x}_{\text{moon}}$$



Hierarchical Transformation

- Sun – Earth
 - The earth is orbiting around the sun

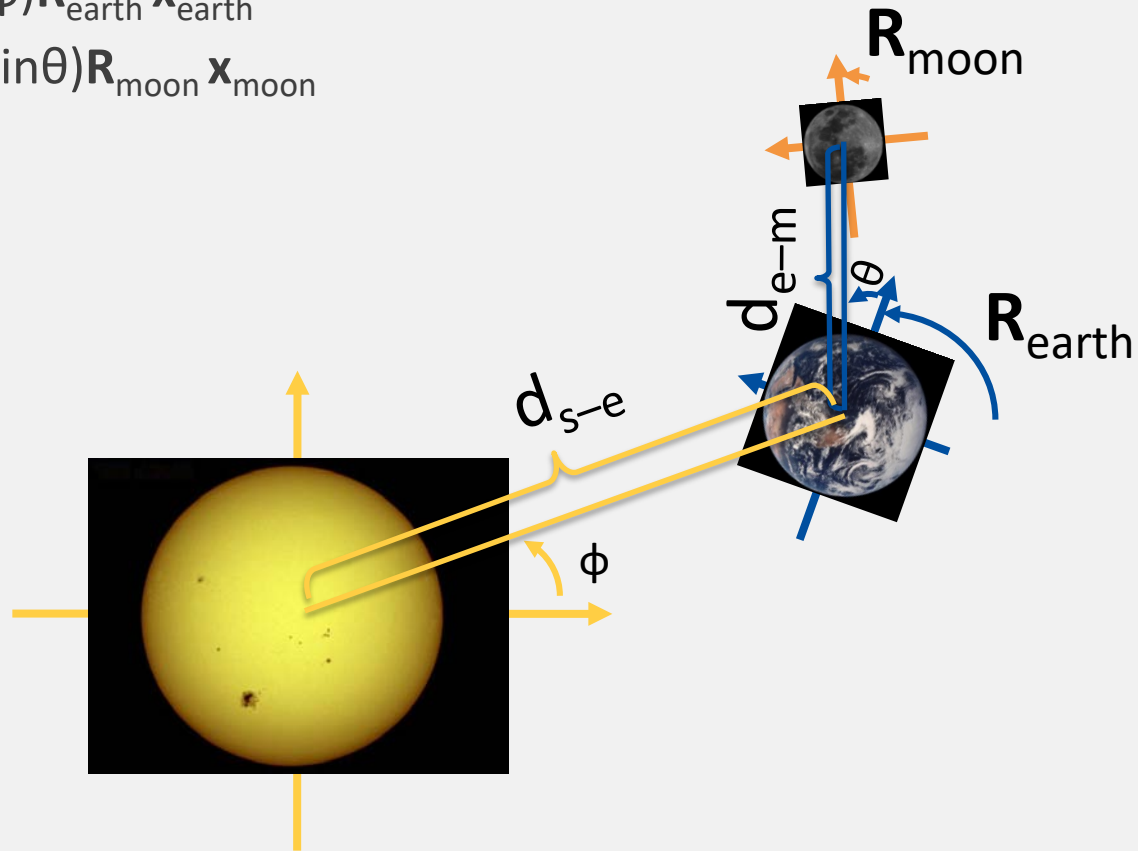
$$\mathbf{x}_{\text{sun}} = \mathbf{T}_{\text{earth}}(d_{s-e} \cos \phi, d_{s-e} \sin \phi) \mathbf{R}_{\text{earth}} \mathbf{x}_{\text{earth}}$$



Hierarchical Transformation

- Sun – Earth – Moon

- $\mathbf{x}_{\text{sun}} = \mathbf{T}_{\text{earth}}(d_{s-e} \cos \phi, d_{s-e} \sin \phi) \mathbf{R}_{\text{earth}} \mathbf{x}_{\text{earth}}$
- $\mathbf{x}_{\text{earth}} = \mathbf{T}_{\text{moon}}(d_{e-m} \cos \theta, d_{e-m} \sin \theta) \mathbf{R}_{\text{moon}} \mathbf{x}_{\text{moon}}$



Hierarchical Transformation

- Sun – Earth – Moon

- $\mathbf{x}_{\text{sun}} = \mathbf{T}_{\text{earth}}(d_{\text{s-e}} \cos \phi, d_{\text{s-e}} \sin \phi) \mathbf{R}_{\text{earth}} \mathbf{x}_{\text{earth}}$

- $\mathbf{x}_{\text{earth}} = \mathbf{T}_{\text{moon}}(d_{\text{e-m}} \cos \theta, d_{\text{e-m}} \sin \theta) \mathbf{R}_{\text{moon}} \mathbf{x}_{\text{moon}}$

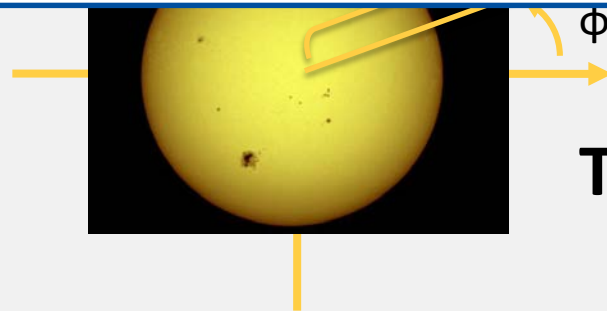


Transformation of the Earth w.r.t. the frame of the Sun

$$\mathbf{x}_{\text{sun}} = \mathbf{T}_{\text{earth}}(d_{\text{s-e}} \cos \phi, d_{\text{s-e}} \sin \phi) \mathbf{R}_{\text{earth}} \mathbf{x}_{\text{earth}}$$

Transformation of the Moon w.r.t. the frame of the Sun

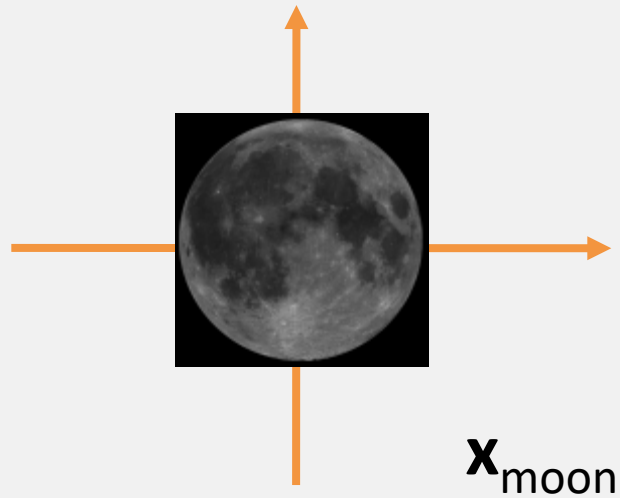
$$\mathbf{x}_{\text{sun}} = \mathbf{T}_{\text{earth}}(d_{\text{s-e}} \cos \phi, d_{\text{s-e}} \sin \phi) \mathbf{R}_{\text{earth}} \mathbf{T}_{\text{moon}}(d_{\text{e-m}} \cos \theta, d_{\text{e-m}} \sin \theta) \mathbf{R}_{\text{moon}} \mathbf{x}_{\text{moon}}$$



$$\mathbf{T}_{\text{earth}}(d_{\text{s-e}} \cos \phi, d_{\text{s-e}} \sin \phi)$$

OpenGL ES 1.x Codes – Hierarchical Transformation

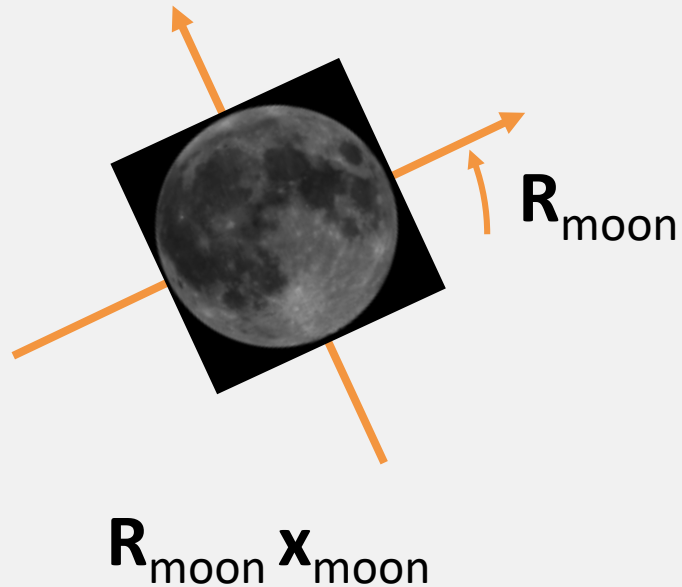
- Moon
 - Modeling the moon



```
void draw_moon()
{
    glEnableClientState(GL_VERTEX_ARRAY);
    glEnableClientState(GL_COLOR_ARRAY);
    glVertexPointer(...);
    glColorPointer(...);
    glDrawElements(...);
    glDisableClientState(GL_VERTEX_ARRAY);
    glDisableClientState(GL_COLOR_ARRAY);
}
```


OpenGL ES 1.x Codes – Hierarchical Transformation

- Moon
 - Modeling the moon
 - Rotating itself



```
void draw_earth_system()
{
    // ...
    glRotatef(...);    // rotating the Moon
    draw_moon();
}

void draw_moon() { // ... glDrawElements(); ... }
```

OpenGL ES 1.x Codes – Hierarchical Transformation

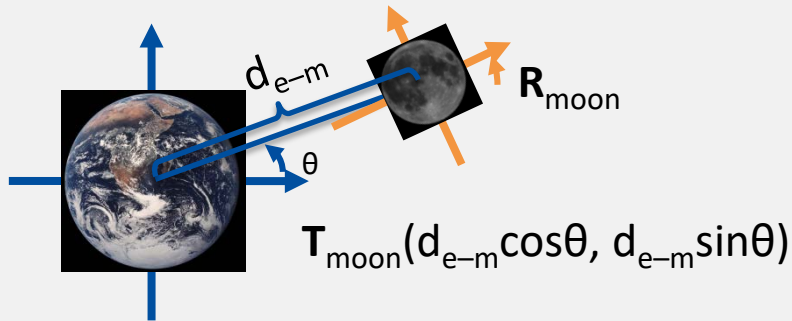
- Earth – Moon
 - The moon is orbiting around the earth

$$\mathbf{x}_{\text{earth}} = \mathbf{T}_{\text{moon}}(d_{\text{e-m}} \cos \theta, d_{\text{e-m}} \sin \theta) \mathbf{R}_{\text{moon}} \mathbf{x}_{\text{moon}}$$

```
void draw_earth_system()
{
    draw_earth();

    glTranslatef(...); // orbiting around the Earth
    glRotatef(...);    // rotating the Moon
    draw_moon();
}

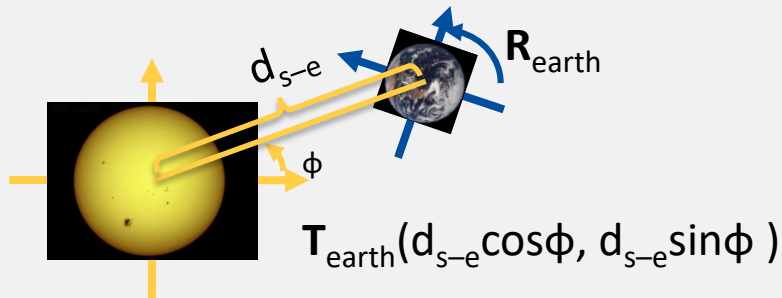
void draw_moon() { // ... glDrawElements(); ... }
void draw_earth() { // ... glDrawElements(); ... }
```



OpenGL ES 1.x Codes – Hierarchical Transformation

- Sun – Earth
 - The earth is orbiting around the sun

$$\mathbf{x}_{\text{sun}} = \mathbf{T}_{\text{earth}}(d_{s-e} \cos \phi, d_{s-e} \sin \phi) \mathbf{R}_{\text{earth}} \mathbf{x}_{\text{earth}}$$



```
void draw_sun_system()
{
    draw_sun();

    glTranslatef(...); // orbiting around the Sun
    glRotatef(...);    // rotating the Earth system
    draw_earth_system();
}

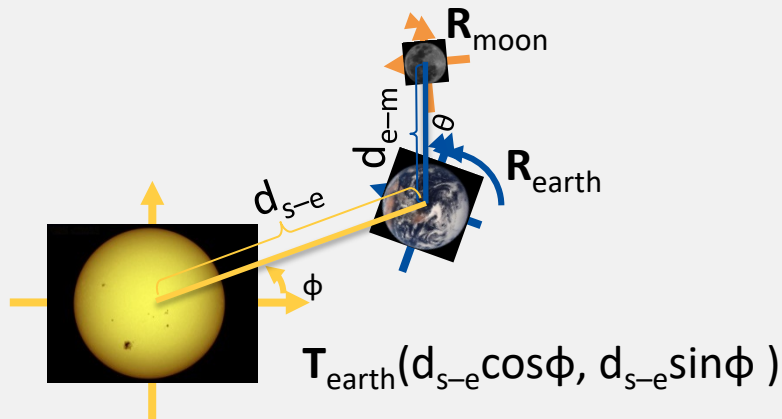
void draw_moon() { // ... glDrawElements(); ... }
void draw_earth() { // ... glDrawElements(); ... }
void draw_sun() { // ... glDrawElements(); ... }
```

OpenGL ES 1.x Codes – Hierarchical Transformation

- Sun – Earth – Moon

$$\mathbf{x}_{\text{earth}} = \mathbf{T}_{\text{moon}}(d_{\text{e-m}} \cos \theta, d_{\text{e-m}} \sin \theta) \mathbf{R}_{\text{moon}} \mathbf{x}_{\text{moon}}$$

$$\mathbf{x}_{\text{sun}} = \mathbf{T}_{\text{earth}}(d_{\text{s-e}} \cos \phi, d_{\text{s-e}} \sin \phi) \mathbf{R}_{\text{earth}} \mathbf{x}_{\text{earth}}$$



```
void draw_sun_system()
{
    draw_sun();

    glTranslatef(...); // orbiting around the Sun
    glRotatef(...);    // rotating the Earth system
    draw_earth_system();
}

void draw_earth_system()
{
    draw_earth();

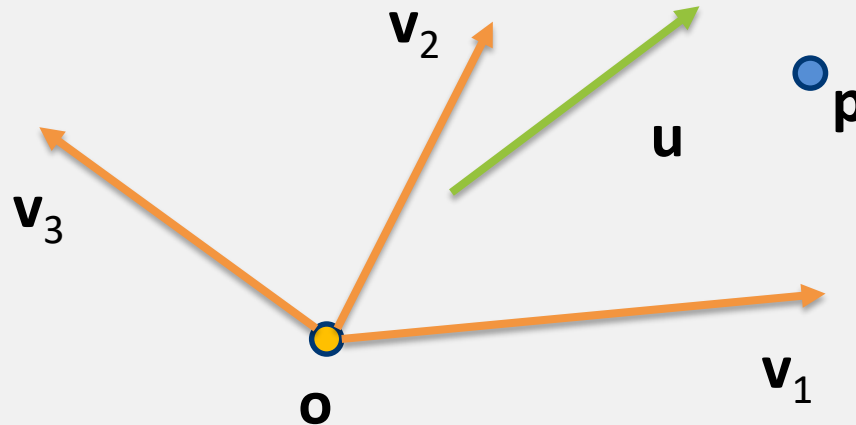
    glTranslatef(...); // orbiting around the Earth
    glRotatef(...);    // rotating the Moon
    draw_moon();
}

void draw_moon() { // ... glDrawElements(); ... }
void draw_earth() { // ... glDrawElements(); ... }
void draw_sun() { // ... glDrawElements(); ... }
```

View Transformations

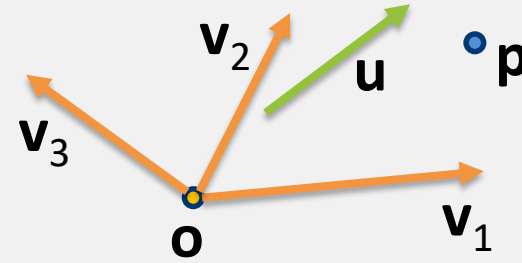
Coordinate System (Frame)

- A Coordinate system (or frame) consists of a set of basis vectors and an origin
 - A set of basis vectors: $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$
 - An origin: \mathbf{o}
- How to representing a vector \mathbf{u} and a point \mathbf{p} w.r.t. a given coordinate system?



Coordinate System (Frame)

- Consider a set of basis vectors and an origin
 - $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$
 - \mathbf{o}
- A vector \mathbf{u} is written
 - $\mathbf{u} = \alpha_1 \mathbf{v}_1 + \alpha_2 \mathbf{v}_2 + \dots + \alpha_n \mathbf{v}_n + 0 \cdot \mathbf{o}$
 - The list of scalars, $\{\alpha_1, \alpha_2, \dots, \alpha_n, 0\}$ is the representation of \mathbf{u} w.r.t. the given coordinate system
- A point \mathbf{p} is written
 - $\mathbf{p} = \beta_1 \mathbf{v}_1 + \beta_2 \mathbf{v}_2 + \dots + \beta_n \mathbf{v}_n + 1 \cdot \mathbf{o}$
 - The list of scalars, $\{\beta_1, \beta_2, \dots, \beta_n, 1\}$ is the representation of \mathbf{p} w.r.t. the given coordinate system

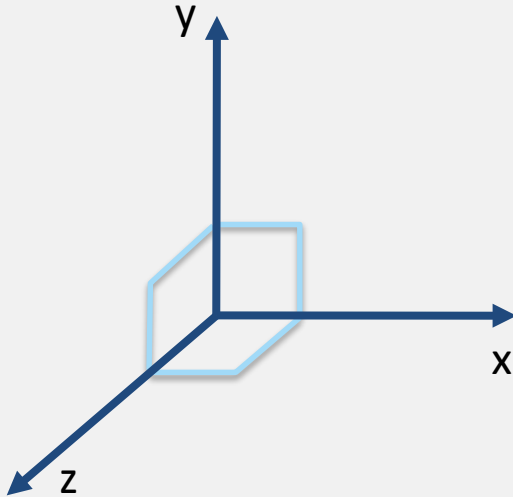


$$\mathbf{u} = [\alpha_1 \quad \alpha_2 \quad \cdots \quad \alpha_n]^T = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix}$$

$$\mathbf{p} = [\beta_1 \quad \beta_2 \quad \cdots \quad \beta_n]^T = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{bmatrix}$$

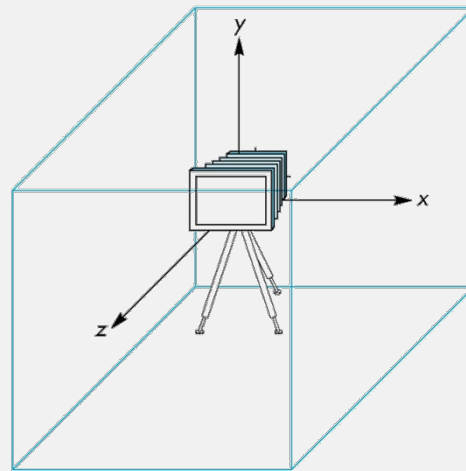
Coordinate System (Frame)

- In OpenGL ES, we just care about **orthonormal** frames
 - **Ortho** means that the basis vectors are orthogonal to each other
 - $x\text{-axis} \perp y\text{-axis} \perp z\text{-axis}$
 - **normal** means that the length of each basis vector is 1
 - The unit length of each axis is equal to 1



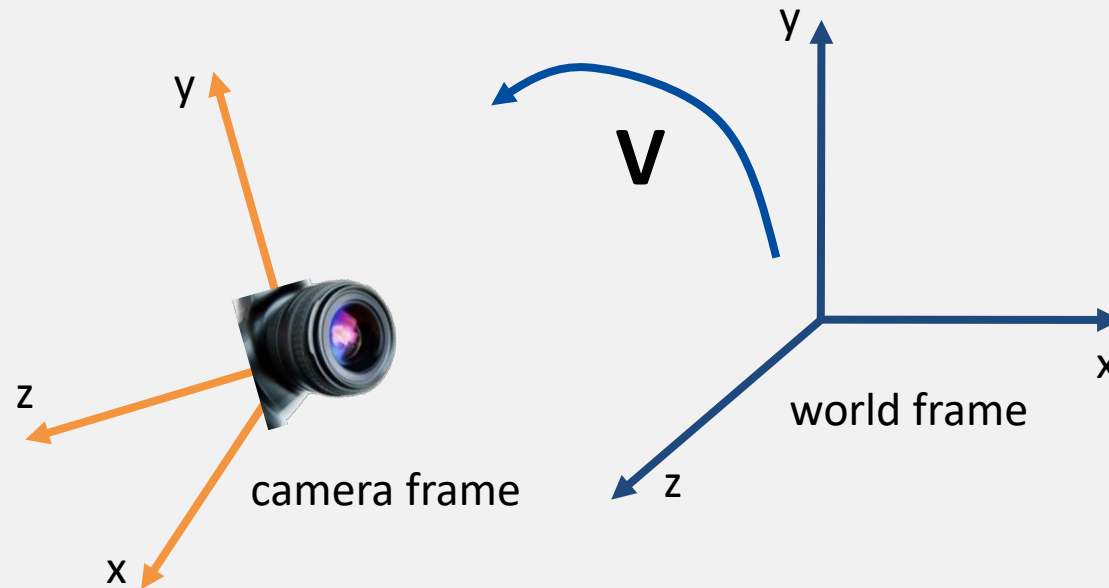
Inside of gluLookAt()

- Initially, OpenGL ES camera coordinate is as follows
 - Center of projection (COP) is placed in the origin
 - Right direction is the positive direction of x-axis
 - Up direction is the positive direction of y-axis
 - Viewing direction is the negative direction of z-axis



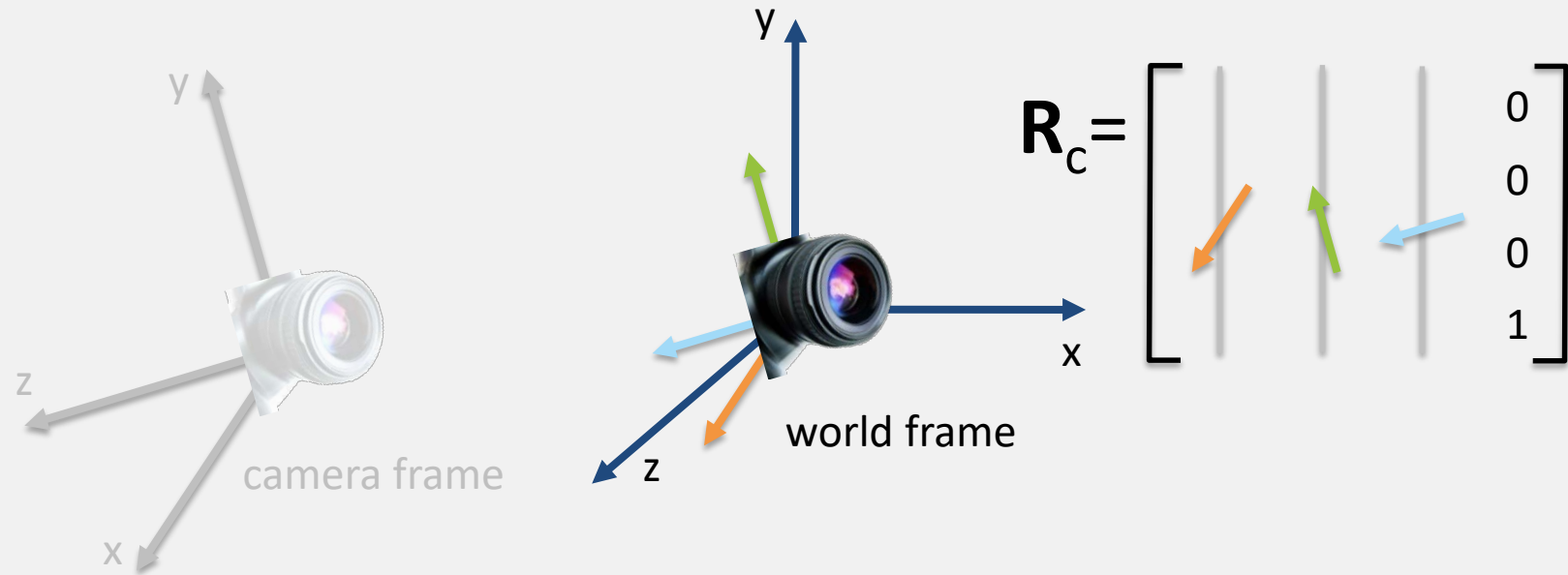
Inside of gluLookAt()

- When we apply gluLookAt()
 - Rotate the camera frame on the world frame: \mathbf{R}_c
 - Translate the camera frame on the world frame: \mathbf{T}_c
 - Therefore, $\mathbf{V} = \mathbf{T}_c \mathbf{R}_c$ and gluLookAt() generates $\mathbf{V}^{-1} = \mathbf{R}_c^{-1} \mathbf{T}_c^{-1}$



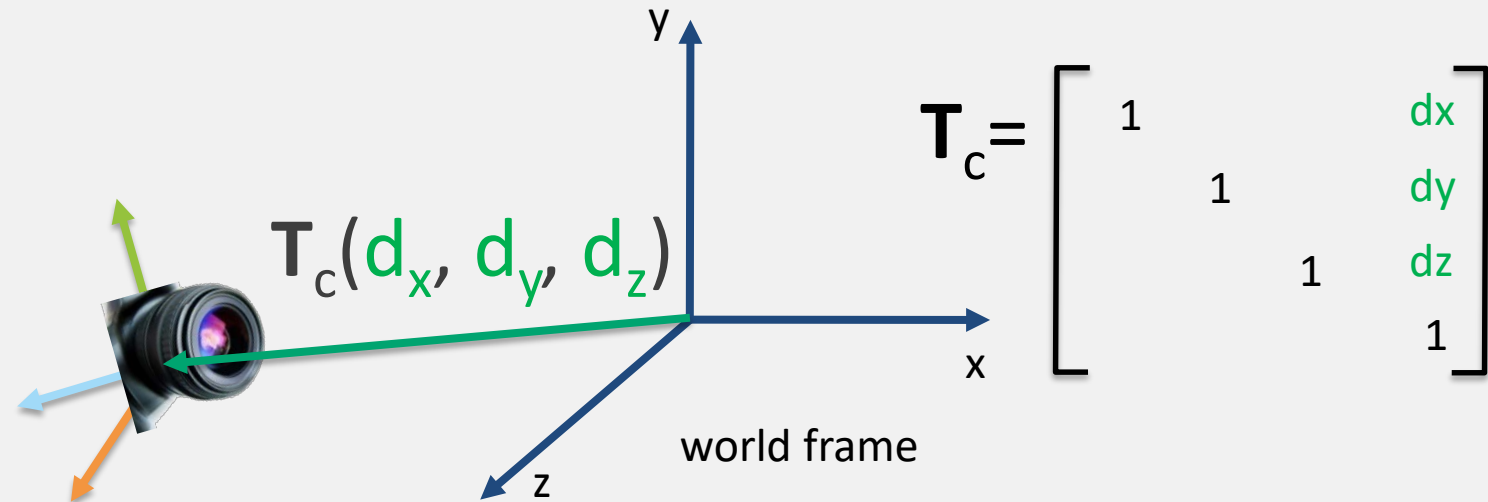
Inside of gluLookAt()

- When we apply gluLookAt()
 - Rotate the camera frame on the world frame: \mathbf{R}_c
 - Translate the camera frame on the world frame: \mathbf{T}_c
 - Therefore, $\mathbf{V} = \mathbf{T}_c \mathbf{R}_c$ and gluLookAt() generates $\mathbf{V}^{-1} = \mathbf{R}_c^{-1} \mathbf{T}_c^{-1}$



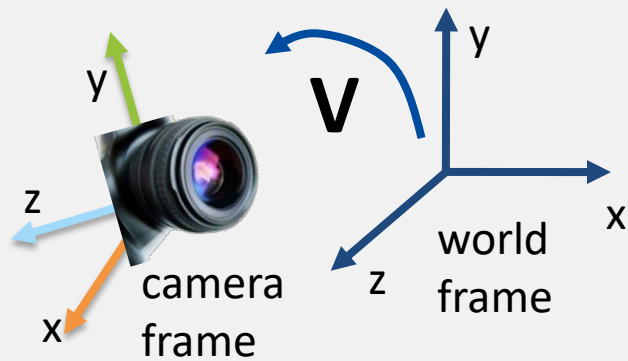
Inside of gluLookAt()

- When we apply gluLookAt()
 - Rotate the camera frame on the world frame: \mathbf{R}_c
 - Translate the camera frame on the world frame: \mathbf{T}_c
 - Therefore, $\mathbf{V} = \mathbf{T}_c \mathbf{R}_c$ and gluLookAt() generates $\mathbf{V}^{-1} = \mathbf{R}_c^{-1} \mathbf{T}_c^{-1}$



Inside of gluLookAt()

- When we apply gluLookAt()
 - Rotate the camera frame on the world frame: \mathbf{R}_c
 - Translate the camera frame on the world frame: \mathbf{T}_c
 - Therefore, $\mathbf{V} = \mathbf{T}_c \mathbf{R}_c$ and gluLookAt() generates $\mathbf{V}^{-1} = \mathbf{R}_c^{-1} \mathbf{T}_c^{-1}$

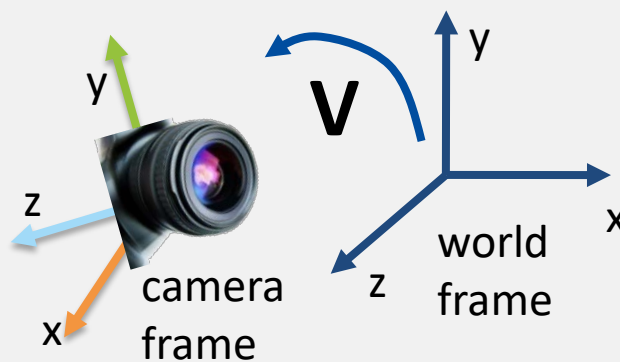


$$\mathbf{V} = \mathbf{T}_c \mathbf{R}_c$$
$$= \begin{bmatrix} 1 & & & dx \\ & 1 & & dy \\ & & 1 & dz \\ & & & 1 \end{bmatrix} \begin{bmatrix} | & | & | & 0 \\ | & | & | & 0 \\ | & | & | & 0 \\ | & | & | & 1 \end{bmatrix}$$

The diagram illustrates the transformation matrix \mathbf{V} as the product of a translation matrix \mathbf{T}_c and a rotation matrix \mathbf{R}_c . The translation matrix \mathbf{T}_c is shown as a 4x4 matrix with 1s on the diagonal and translation components dx , dy , and dz in the fourth column. The rotation matrix \mathbf{R}_c is represented by three vertical gray lines, with colored arrows (orange, green, blue) indicating the rotation of the camera frame axes.

Inside of gluLookAt()

- When we apply gluLookAt()
 - Rotate the camera frame on the world frame: \mathbf{R}_c
 - Translate the camera frame on the world frame: \mathbf{T}_c
 - Therefore, $\mathbf{V} = \mathbf{T}_c \mathbf{R}_c$ and gluLookAt() generates $\mathbf{V}^{-1} = \mathbf{R}_c^{-1} \mathbf{T}_c^{-1}$



$\mathbf{V}^{-1} = \mathbf{R}_c^{-1} \mathbf{T}_c^{-1}$

$$= \begin{bmatrix} \text{---} & \text{---} & \text{---} \\ \text{---} & \text{---} & \text{---} \\ \text{---} & \text{---} & \text{---} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{bmatrix} \begin{bmatrix} -dx \\ -dy \\ -dz \\ 1 \end{bmatrix}$$

Revisiting Linear Transformations

Translation (이동)

- Translation matrix **T**

- Translation can also be expressed by using a 4x4 matrix **T** in homogeneous coordinates

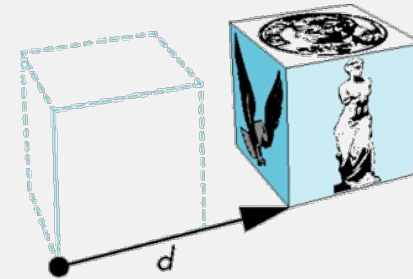
- $\mathbf{x}' = \mathbf{T}\mathbf{x}$

- $\mathbf{T} = \mathbf{T}(d_x, d_y, d_z) =$

$$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- [`glTranslate`](#)(d_x, d_y, d_z)

- This form is better for H/W implementation



Translation (이동) 행렬의 해석: Homogeneous Coordinate 활용

$$\mathbf{T}(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{I}_{3 \times 3} & \mathbf{d} \\ \mathbf{0}^T & 1 \end{bmatrix} = \mathbf{T}(\mathbf{d}) = \mathbf{T}_d$$

- 단, $\mathbf{d} = (d_x, d_y, d_z)$.

Translation about Points or Vectors

Translation about Points

- 위치 $\mathbf{p} = (p_x, p_y, p_z)$ 를 $\mathbf{d} = (d_x, d_y, d_z)$ 만큼 이동

- Homogeneous coordinate 이용

$$\mathbf{T}_d \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{I}_{3 \times 3} & \mathbf{d} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{p} + \mathbf{d} \\ 1 \end{bmatrix}$$

- Homogeneous coordinate 이용치 않은 경우

- $\mathbf{p} + \mathbf{d}$

Translation about Vectors

- 벡터 $\mathbf{v} = (v_x, v_y, v_z)$ 를 $\mathbf{d} = (d_x, d_y, d_z)$ 만큼 이동

- Homogeneous coordinate 이용한 경우

$$\mathbf{T}_d \begin{bmatrix} \mathbf{v} \\ 0 \end{bmatrix} = \begin{bmatrix} \mathbf{I}_{3 \times 3} & \mathbf{d} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ 0 \end{bmatrix} = \begin{bmatrix} \mathbf{v} \\ 0 \end{bmatrix}$$

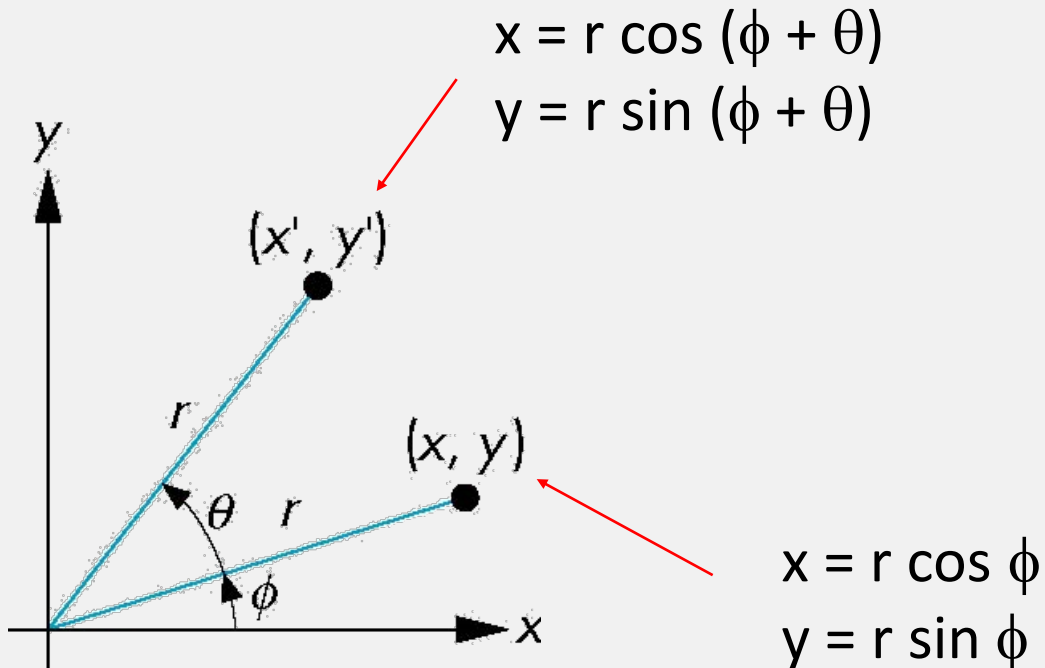
- Homogeneous coordinate 이용치 않은 경우

- \mathbf{v}

- 즉, 벡터 \mathbf{v} 는 이동 \mathbf{d} 에 영향이 없음

Rotation (회전)

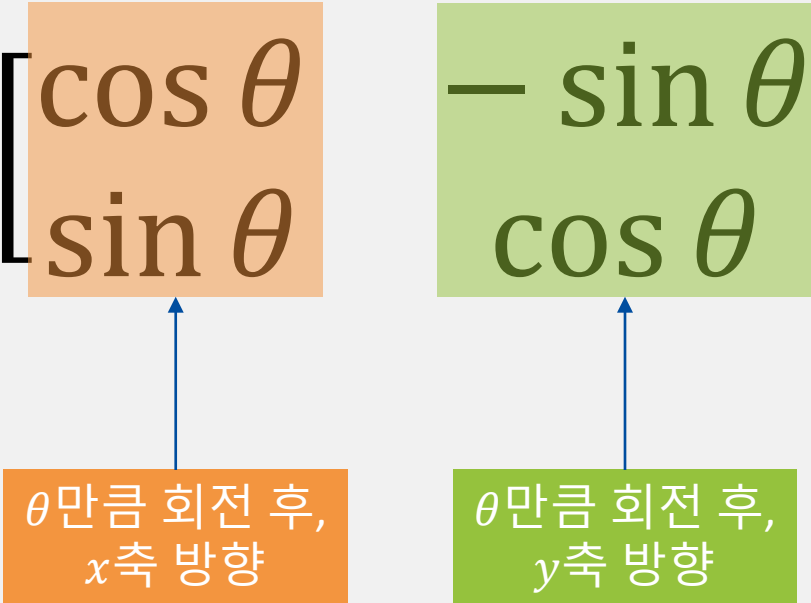
- Consider rotation about the origin by θ degrees
 - Radius stays the same, angle increases by θ



$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Rotation (회전) 행렬의 해석

- 2차원 회전 행렬 해석
 - 고교과정에서 배운 내용에 대한 재해석

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$


θ 만큼 회전 후,
 x 축 방향

θ 만큼 회전 후,
 y 축 방향

Rotation (회전) 행렬의 해석: Homogeneous Coordinate 활용

- 2차원 회전 행렬 해석
 - Homogeneous coordinate를 이용한 해석

$$\mathbf{R}(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}(\theta)_{2 \times 2} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

θ 만큼 회전 후, x 축 방향

θ 만큼 회전 후, y 축 방향

원점

Rotation (회전) 행렬의 해석: Homogeneous Coordinate 활용

2D Rotation of Points

- 위치 $\mathbf{p} = (p_x, p_y)$ 를 θ 만큼 회전
 - Homogeneous coordinate를 이용한 해석

$$\mathbf{R}(\theta) \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}(\theta)_{2 \times 2} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}(\theta)_{2 \times 2} \mathbf{p} \\ 1 \end{bmatrix}$$

- Homogeneous coordinate 이용치 않은 경우
 - $\mathbf{R}(\theta)_{2 \times 2} \mathbf{p}$

2D Rotation of Vectors

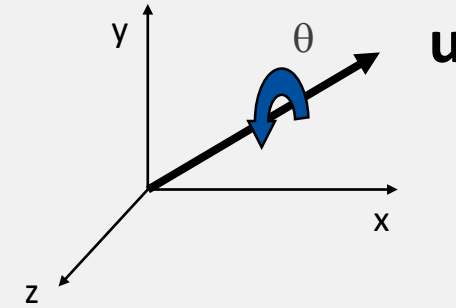
- 벡터 $\mathbf{v} = (v_x, v_y)$ 를 θ 만큼 회전
 - Homogeneous coordinate를 이용한 해석

$$\mathbf{R}(\theta) \begin{bmatrix} \mathbf{v} \\ 0 \end{bmatrix} = \begin{bmatrix} \mathbf{R}(\theta)_{2 \times 2} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ 0 \end{bmatrix} = \begin{bmatrix} \mathbf{R}(\theta)_{2 \times 2} \mathbf{v} \\ 0 \end{bmatrix}$$

- Homogeneous coordinate 이용치 않은 경우
 - $\mathbf{R}(\theta)_{2 \times 2} \mathbf{v}$

Rotation (회전)

- General rotation about the **origin**
 - A rotation by θ rotation an arbitrary axis \mathbf{u}
 - [Quaternion](#) can express general rotation
 - $\mathbf{x}' = \mathbf{R}_{\mathbf{u}}(\theta)\mathbf{x}$
 - where,



$$\mathbf{R}_{\mathbf{u}}(\theta) = \begin{bmatrix} \cos \theta + u_x^2(1 - \cos \theta) & u_x u_y(1 - \cos \theta) - u_z \sin \theta & u_x u_z(1 - \cos \theta) + u_y \sin \theta & 0 \\ u_y u_x(1 - \cos \theta) + u_z \sin \theta & \cos \theta + u_y^2(1 - \cos \theta) & u_y u_z(1 - \cos \theta) - u_x \sin \theta & 0 \\ u_z u_x(1 - \cos \theta) - u_y \sin \theta & u_z u_y(1 - \cos \theta) + u_x \sin \theta & \cos \theta + u_z^2(1 - \cos \theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}_{\mathbf{u}}(\theta)_{3 \times 3} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

↑

u축을 중심으로
 θ 만큼 회전 후,
 x 축 방향

↑

u축을 중심으로
 θ 만큼 회전 후,
 y 축 방향

↑

u축을 중심으로
 θ 만큼 회전 후,
 z 축 방향

↑

원점

- [glRotate](#)(θ, u_x, u_y, u_z), where $\mathbf{u} = (u_x, u_y, u_z)$

Rotation (회전) 행렬의 해석: Homogeneous Coordinate 활용

3D Rotation of Points

- 위치 $\mathbf{p} = (p_x, p_y, p_z)$ 를 \mathbf{u} 축을 중심으로 θ 만큼 회전
 - Homogeneous coordinate를 이용한 해석

$$\mathbf{R}_{\mathbf{u}}(\theta) \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}_{\mathbf{u}}(\theta)_{3 \times 3} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}_{\mathbf{u}}(\theta)_{3 \times 3} \mathbf{p} \\ 1 \end{bmatrix}$$

- Homogeneous coordinate 이용치 않은 경우
 - $\mathbf{R}_{\mathbf{u}}(\theta)_{3 \times 3} \mathbf{p}$

3D Rotation of Vectors

- 벡터 $\mathbf{v} = (v_x, v_y, v_z)$ 를 \mathbf{u} 축을 중심으로 θ 만큼 회전
 - Homogeneous coordinate를 이용한 해석

$$\mathbf{R}_{\mathbf{u}}(\theta) \begin{bmatrix} \mathbf{v} \\ 0 \end{bmatrix} = \begin{bmatrix} \mathbf{R}_{\mathbf{u}}(\theta)_{3 \times 3} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ 0 \end{bmatrix} = \begin{bmatrix} \mathbf{R}_{\mathbf{u}}(\theta)_{3 \times 3} \mathbf{v} \\ 0 \end{bmatrix}$$

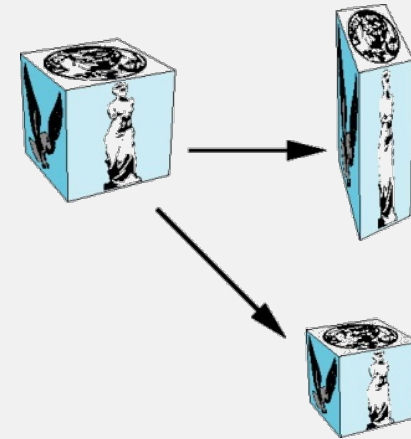
- Homogeneous coordinate 이용치 않은 경우
 - $\mathbf{R}_{\mathbf{u}}(\theta)_{3 \times 3} \mathbf{v}$

Scaling (확대 축소)

- Scaling matrix **S**
 - Expand or contract along each axis (fixed point of **origin**)
 - $\mathbf{x}' = \mathbf{S}\mathbf{x}$

- $\mathbf{S} = \mathbf{S}(s_x, s_y, s_z) =$

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



- [`glScale`](#)(s_x, s_y, s_z)

Scale (확대축소) 행렬의 해석: Homogeneous Coordinate 활용

$$\mathbf{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{S}_{3 \times 3} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} = \mathbf{S}(\mathbf{s}) = \mathbf{S}_s$$

- 단, $\mathbf{s} = (s_x, s_y, s_z)$.

Revisiting Composite Transformation

Composite Transformation (합성 변환)

- We can composite transformation by multiplying matrices of rotation, translation, and scaling.

– Example:

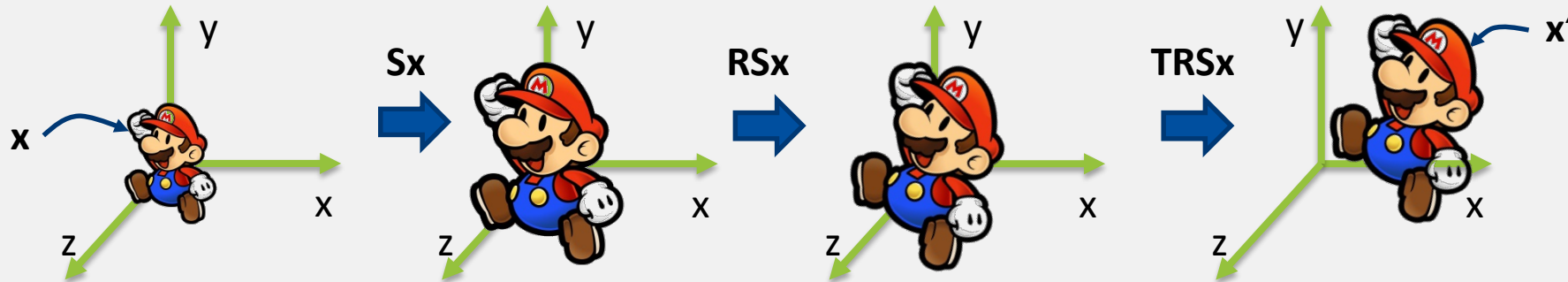
- 1) Uniformly scale 2x:
- 2) Rotate -30 degrees by +z axis:
- 3) Translate by (2, 1, 0):

- $\mathbf{x}' = \mathbf{T}(\mathbf{R}(\mathbf{S}\mathbf{x})) = \mathbf{TRSx}$

– \mathbf{x} : original object

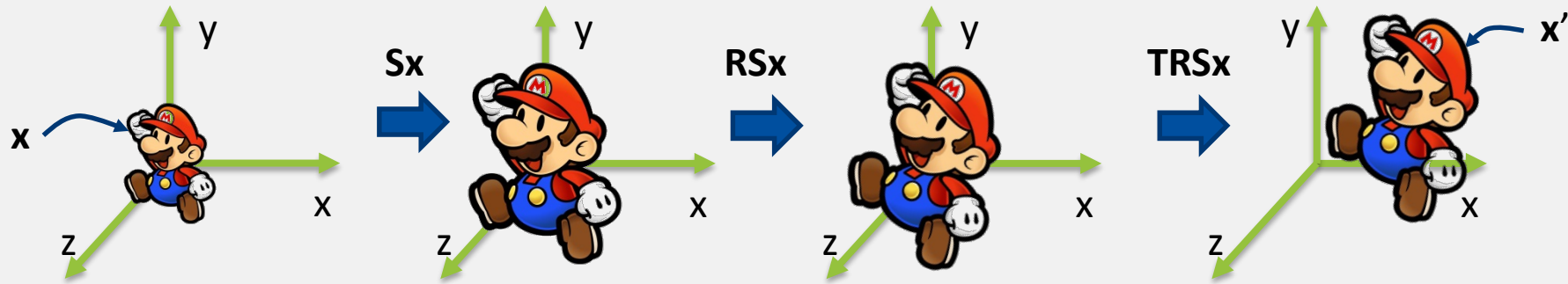
– \mathbf{x}' : transformed object

```
glTranslatef(2, 1, 0);      // T
glRotatef(-30, 0, 0, 1);   // R
glScalef(2, 2, 2);         // S
glDrawElements(...);      // x
```



Composite Transformation (합성 변환)

- Composite transformation w/ “scale, rotation, and then translation”



$$\mathbf{T}_d \mathbf{R}_u(\theta) \mathbf{S}_s = \begin{bmatrix} \mathbf{I}_{3 \times 3} & \mathbf{d} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R}_u(\theta)_{3 \times 3} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{S}_{3 \times 3} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{SR}_u(\theta)_{3 \times 3} & \mathbf{d} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

Inside of gluLookAt()

- When we apply gluLookAt()
 - Rotate the camera frame on the world frame: \mathbf{R}_c
 - Translate the camera frame on the world frame: \mathbf{T}_c
 - Therefore, $\mathbf{V} = \mathbf{T}_c \mathbf{R}_c$ and gluLookAt() generates $\mathbf{V}^{-1} = \mathbf{R}_c^{-1} \mathbf{T}_c^{-1}$

